

phy++ v1.0

Contents

1	Introduction	5
1.1	Getting started	5
2	Core library	6
2.1	Overview	6
2.1.1	Operator overloading	7
2.1.2	Multi-dimensional indexing	8
2.1.3	Vector views	9
2.2	The vector class	11
2.2.1	Member variables	11
2.2.2	Constructors and assignment	12
2.2.3	Member functions	15
2.2.4	Indexing	20
2.2.5	Operators	24
2.3	The view class	24
2.3.1	Member functions	25
2.4	Metaprogramming traits and functions (WIP)	25
3	Support libraries	26
3.1	Generic vector functions	30
3.1.1	Range-based iteration	30
3.1.2	Index manipulation	31
3.1.3	Integer sequences	32
3.1.4	Rearranging elements	32
3.1.5	Finding elements	34
3.1.6	Modifying dimensions	38
3.1.7	Adding/removing elements	40
3.1.8	Error checking	41

3.2	Parsing command line arguments	42
3.3	File input/output	42
3.4	String manipulation	43
3.5	OS interaction	44
3.6	Printing to the terminal	44
3.7	Measuring time	45
3.8	Mathematics	45
3.8.1	Low level mathematics	46
3.8.2	Sequences and bins	47
3.8.3	Randomization	47
3.8.4	Reduction	48
3.8.5	Interpolation	49
3.8.6	Calculus	49
3.8.7	Algebra	50
3.8.8	Fitting	51
3.8.9	Geometry	51
3.8.10	Debug functions	52
3.9	Parallel execution	52
3.10	FITS input/output	52
3.10.1	Generic header functions	53
3.10.2	FITS images	53
3.10.3	WCS coordinates	54
3.10.4	FITS tables	54
3.11	Image processing	54
3.12	Astrophysics	55
3.12.1	PSF fitting	55
3.12.2	Cosmology	55
3.12.3	Fluxes, magnitudes and luminosities	55
3.12.4	Sky coordinates	56
3.12.5	Catalog management	57
3.12.6	Image stacking	57
3.12.7	Template fitting	57
4	Tools	58
4.1	Astrophysics	59
4.1.1	angcorrel	59
4.1.2	catinfo	59
4.1.3	deg2sex and sex2deg	59

4.1.4	findsrc	59
4.1.5	fluxcube	59
4.1.6	getgal	59
4.1.7	photinfo	59
4.1.8	psffit	59
4.1.9	qstack2	59
4.1.10	qxmatch2	59
4.1.11	randsrc	59
4.1.12	subsrc	59
4.2	FITS and ASCII	59
4.2.1	fits2ascii	59
4.2.2	fitstool	59
4.2.3	imgtool	59
4.2.4	qconvol	59
4.2.5	remcol	59
	Index	60

Chapter 1

Introduction

1.1 Getting started

Chapter 2

Core library

2.1 Overview

At the core of the `phy++` library is the *vector* class. This is basically an enhanced `std::vector`¹, and it therefore shares most of its features and strengths. In particular, a vector can contain zero, one, or as many elements as your computer can handle. Its size is defined at *run-time*, meaning that its content can vary depending on user input, but also that a vector can change its total number of elements at any time. These elements are stored contiguously in memory, which provides optimal performances in most situations. Lastly, a vector is an homogeneous container, meaning that a given vector can only contain a single type of elements (e.g., `int` or `float`, but not both at the same time).

Like most advanced C++ libraries, `phy++` is essentially *template* based. This means that most of the code is written to work for *any* type `T`, e.g., `int`, `float`, `std::string`, or whatever you need. However, while templates are a fantastic tool for library writers, they can easily become a burden for the *user* of the library. The good thing is, since `phy++` is a numerical analysis library, we know in advance what types will most often be stored inside the vectors. So for this reason, to reduce typing and enhance readability, we introduce type aliases for the most commonly used vector types:

- `vec1f`: vector of `float`,
- `vec1d`: vector of `double`,
- `vec1cf`: vector of `std::complex<float>`,
- `vec1cd`: vector of `std::complex<double>`,

¹In fact, `std::vector` is used to implement the `phy++` vectors internally.

- vec1i: vector of **int** (precisely, **int_t** = **std::ptrdiff_t**),
- vec1u: vector of **unsigned int** (precisely, **uint_t** = **std::size_t**),
- vec1b: vector of **bool**,
- vec1s: vector of **std::string**,
- vec1c: vector of **char**.

On top of the **std::vector** interface, the **phy++** vector adds some extra functionalities. The most important ones are operator overloading, multi-dimensional indexing, and vector views.

2.1.1 Operator overloading

The only thing you can do to operate on all the elements of an **std::vector** is to iterate over these elements explicitly, either using a C++11 range-based loop, or using indices:

```
// Goal: multiply all elements by two.
std::vector<float> v = {1,2,3,4};

// Either using a range-based loop,
for (float& x : v) {
    x *= 2;
}

// ... or an index-based loop.
for (std::size_t i = 0; i < v.size(); ++i) {
    v[i] *= 2;
}
```

While this is fairly readable (especially the first version), it is still not very concise and expressive. In **phy++**, we have *overloaded* the usual mathematical operators on our vector type, meaning that it is possible to write the above code in a much simpler way:

```
// Using phy++ vector.
vec1f w = {1,2,3,4};
w *= 2; // {2,4,6,8}
```

Not only this, but we can perform operations on a pair of vectors in the same way:

```
// Goal: sum the content of the two vectors.
vec1f x = {1,2,3,4}, y = {4,3,2,1};
vec1f z = x + y; // {5,5,5,5}
```

Warning

The only issue with operator overloading concerns the hat operator `^`. In most languages, this operator is used for exponentiation, e.g., `4^2 == 16`. However, in C++ this value is actually 6, because the hat operator is the binary XOR (exclusive-or). Even worse, the hat operator in C++ does not have the same *precedence* as in regular mathematics: it has a lower priority than any other mathematical operator. Both these reasons make it unwise to overload the hat operator in `phy++`. In order to perform exponentiation, you will have to use a dedicated function such as `pow(4,2)` (3.8).

2.1.2 Multi-dimensional indexing

The standard `std::vector` is a purely linear container: one can access its elements using `v[i]`, with `i` ranging from 0 up to `std::vector::size()-1` (included). However, the `phy++` vector allows N-dimensional indexing, i.e., using a group of indices to identify one element. For example, this is particularly useful to work on images, which are essentially 2-dimensional objects where one identifies a given pixel by its coordinates `x` and `y`. The natural syntax for this indexing would be to write `img[x,y]`. This syntax is valid C++ code, but unfortunately will not do what you expect² and there is no sane way around it. The alternative we chose here is to write instead `img(x,y)`. While it is not as semantically clear as using brackets, it has the nice advantage of being compatible with the IDL syntax.

The multi-dimensional nature of a vector is determined at *compile time*, i.e., it cannot be changed after the vector is declared. By default, a vector is mono-dimensional. To use the above feature, one needs to specify the number of needed dimensions in the type of the vector. For example, a 2D image of `float` will be declared as `vec2f`. These type aliases are provided for dimensions up to 6. Here is an example of manipulation of a 2D matrix:

```
// Create a simple matrix.
vec2f m = {{1,2,3}, {4,5,6}, {7,8,9}};

// Index ordering is similar to C arrays: the last index
// is contiguous in memory. Note that this is *opposite*
```

²This will call the *comma* operator, which evaluates both elements `x` and `y` and returns the last one, i.e., `y`. So this code actually means `img[y]`. With proper configuration, most compiler will warn about this though, since in this context `x` is a useless statement, so you should be safe should you make this mistake.


```
// to the IDL convention.
m(0,0); // 1
m(0,1); // 2
m(1,0); // 4

// It is still possible to access elements as if in a
// "flat" vector
m[0]; // 1
m[1]; // 2
m[3]; // 4
```

Advanced

If for some reason you need to use more than 6 dimensions, or if you need to declare a vector of some type which is not covered above, you can always fall back to the full template syntax:

```
// Need 12 dimensions?
using vec12f = vec_t<12,float>;
// Need 512 bit of floating point precision?
using vec3f512 = vec_t<3,mpfr::real<512>>;
```

The hard limit on the number of dimensions will then depend on your compiler (each dimension involves an additional level of template recursion). The C++ standard does not guarantee anything, but you should be able to go as high as 256 on all major compilers. Beyond this, you should probably see a doctor first.

As for the types allowed inside the vector, there is no explicit restriction. However, some features may obviously not be available depending on the capabilities of your type (e.g., if your type has no **operator***, you will not be able to multiply together vectors of this type). Lastly, the **phy++** vector shares the same restrictions as the `std::vector` regarding the *copyable* and *movable* capabilities of the stored type.

2.1.3 Vector views

We have seen that, instead of accessing each element individually, we can use operator overloading to perform simple operations on all the elements of the vector at once: `w *= 2`. We can also, like with `std::vector`, modify each element individually, knowing their indices: `w[2] *= 2`. One last important feature allowed by the **phy++** vector is that one can create a *view* inside a vector. Each element of the view is actually a *reference* to an

element in the original vector, and modifying the elements of the view actually modifies the elements in the original vector. As you will see later, views actually share most of the interface and capabilities of true vectors, so that most generic codes that work with vectors will also work with views.

```
// Create a simple vector.
vec1f w = {1,2,3,4,5,6};

// We want to "view" only the second, the third and the
// fifth elements. So we first create a vector containing
// the corresponding indices.
vec1u id = {1,2,4};

// Then we create the view.
// Note the usage of "auto" there. The type of the view is
// complex, and it is better not to worry about it.
auto v = w[id];

// Now we can modify these elements very simply, as if they
// were part of a real vector.
v;           // {2,3,5}
v *= 2;      // {4,6,10}
w;           // {1,4,6,4,10,6}
w[1] = 99;   // {1,99,6,4,10,6}
v;           // {99,6,10}
```

Warning

It is important to note that, since a view keeps references to the elements of the original vector, the lifetime of the view must not exceed that of the original vector. Else, it will contain *dangling references*, i.e. pointers to unused memory, and this should be avoided at all cost. In fact, views are not meant to be stored into named variables like in the above example. Most of the time, one will use them as temporary variables, e.g.:

```
vec1f w = {1,2,3,4,5,6};
vec1u id = {1,2,4};
w[id] *= 2; // {1,4,6,4,10,6}
```

2.2 The vector class

The full type of the vector class is

```
template<std::size_t Dimension, typename ElementType>
struct vec_t<Dimension, ElementType>;
```

In the rest of this document, `Dimension` will usually just be called `D` and `ElementType` will be shortened to `T`.

2.2.1 Member variables

A vector only contains two member variables:

- `std::vector<T> data`

This is the underlying `std::vector` containing the elements of the vector. It is exposed to the public interface for simplicity, but on most occasions it should *not* be used directly. In fact, it may become part of the private interface in the future, so you should not rely on its existence.

Advanced

Concerning `bool` vectors. We do not use `std::vector<bool>`, since it is a very special case of `std::vector`: the C++ standard specifies that the `bool` specialization does not store `bool` elements, but is actually implemented like a *bit field*. This is essentially to save memory: a `bool` in C++ occupies 8 bits of memory, like a `char`, even though it can only carry a single bit of information. This is due to *memory alignment* issues, which are inherent to the CPU architecture (the address of individual values in memory are supposed to be multiples of 8 bits, or one byte). In a bit field however, 8 `bool`s are stored in a single `char`, and bitwise operators are used to read and write individual `bool`s. While more memory efficient, it is also slower, and involves a whole machinery to trick the user into thinking that none of this is happening. For this reason, `bool` vectors are implemented with `std::vector<char>` in `phy++`, with one `char` containing one `bool`. This is completely transparent to the library user though, since `char&` is casted into `bool&`, and vice versa, at the boundary of the vector interface.

- `std::array<std::size_t,D> dims`

This variable contains the dimensions of this vector (useful only for multidimensional vectors). On no occasion should you modify this variable yourself: you

should only read its content³. To change the dimensions of a vector, either use `resize(...)` (2.2.3) or assign it another vector (2.2.2).

Example

```
// Create a 2D image with 256 x 128 pixels
vec2f img(256,128);
img.dims[0]; // 256
img.dims[1]; // 128
for (std::size_t x = 0; x < img.dims[0]; ++x)
for (std::size_t y = 0; y < img.dims[1]; ++y) {
    img(x,y) = 3.1415;
}
```

2.2.2 Constructors and assignment

There are various ways to construct a new vector, or assign it some value. In the following, we will cover the various *constructors* and the associated *assignments* when applicable.

- **vec_t::vec_t()**; // *The default constructor*

Like `std::vector`, the `phy++` vector is *default constructible*. By default it is in a valid state where the vector does not contain any element.

Example

```
vec1f w; // default constructor: w is empty
```

- **explicit vec_t::vec_t(...)**; // *The dimension constructor*

The `std::vector` has a constructor to create a new vector of a given size. The `phy++` vector also offers this feature, however it is rendered a bit more complex by the possibility of having multi-dimensional vectors. In particular, it is possible to specify the size either by giving all the dimensions one by one, or by specifying some (or all) as a `std::array`. In all cases, the vector is populated with the number of requested elements, and all these elements are *default initialized* (i.e., integers and floats are initialized to 0, booleans to `false`, and `std::string` are empty). This

³This statement is actually a bit bold, but is mostly true. The only reason why this variable is made public is for optimization purposes. On occasions, it can be noticeably faster to manage manually the growth of a vector, and update the dimensions afterwards. This is often done within the core library, but should rarely be done otherwise.

constructor is declared **explicit** to prevent interference with the other constructors and assignments.

Example

```
vec1f w(10);      // w contains 10 objects all equal to 0
vec2d z(5,4);     // z contains 5*4=20 objects all equal to 0
vec2s x(z.dims);  // x has the same dimensions as z
                  // but contains strings, all empty
vec3b y(z.dims, 8); // y has the same dimensions as z,
                  // plus an extra dimension of length 8
```

- **vec_t::vec_t**(nested_initializer_list<D,T>); // *The list constructor*

Like `std::vector`, the `phy++` vector can be initialized from a list of values. This usually requires the usage of `std::initializer_list<T>`, however here we also have to support multi-dimensional vectors, hence we need initializer lists of initializer lists of ..., and `nested_initializer_list<D,T>` is just that. Note that, since C++ is a row-major language, the most nested lists correspond to the last index in a multi-dimensional vector.

Example

```
vec1f w({1,2,3}); // 1D list constructor: {1,2,3}
vec2f z({{1,2,3},{6,5,4}}); // 2D list constructor: {{1,2,3},{6,5,4}}
z(0,0); // 1
z(0,1); // 2
z(0,2); // 3

// Assignment
w = {4,5,6};
z = {{1,2}, {3,4}, {5,6}}; // dimensions can change through assignment
z(0,0); // 1
z(1,0); // 3
z(2,0); // 5
```

- **vec_t::vec_t**(const vec_t&); // *The copy constructor*

Like `std::vector`, the `phy++` vector is *copiable*, meaning that one can duplicate the content of an existing vector inside another vector by copy. Note that this constructor is only valid for copying vectors of the *same* type. If the type is different, then

another constructor is called (conversion constructor, see below).

Example

```
vec1f w = {1,2,3};  
vec1f z(w); // copy constructor: {1,2,3}  
  
// Assignment  
z = w;
```

- **vec_t::vec_t(vec_t&&);** // The move constructor

Like `std::vector`, the `phy++` vector is *movable*, meaning that one can move the content of an existing vector that is going to be destroyed inside another vector. This is an optimized copy for temporary variables (C++11 move semantics). You need not explicitly ask for either the copy or the move constructor, as they will automatically be chosen by the compiler. For you, this is transparent (but the performance boost is large).

Example

```
vec1f z(vec1f{1,2,3}); // move constructor: {1,2,3}  
// Here, a temporary vector is created with vec1f{1,2,3}.  
// This temporary vector is then *moved* inside z.  
  
// Assignment  
z = vec1f{4,5,6};
```

- **vec_t::vec_t(const vec_t<D,OtherT>&);** // The conversion constructor

C++ supports implicit conversion between all the built-in types. In particular, it is possible to write:

```
int i = 0;  
float f = i; // int to float  
bool b1 = i; // int to bool, mostly for interoperability with C  
bool b2 = f; // float to bool, not sure if that makes sense...
```

While this is very convenient in most cases, it can also lead to dangerous silent conversions, such as the **float** to **bool** conversion. This is, somehow, a legacy of C. In `phy++` we decided to also support such implicit conversions. They make the code much easier to read, but the price to pay is that sometimes we do some conversions

which are not necessary, and we do not realize it because they are implicit. However, we decided to disable implicit conversion to and from **bool** vectors, since it could lead to subtle bugs that are difficult to trace. It is still possible to do the conversion to **bool** using explicit

Example

```
vec1i w = {1,2,0};
vec1f z(w); // int to float {1,2,0}
vec1b y(w); // error: cannot convert int to bool
vec1s x(w); // error: cannot convert int to std::string

vec1b b = {true,true,false};
vec1i s(b); // bool to int {1,1,0}

// Assignment
z = w;
y = w;      // error: cannot convert int to bool
x = w;      // error: cannot convert int to std::string

s = b;      // error: implicit conversion is not allowed for bool
s = vec1i(b); // ok: bool to int {1,1,0}
```

- **vec_t::vec_t(const vec_t<D,T*>&);** // The view constructor

The last constructor on the list is the view constructor. It allows implicit conversion of a view (2.3) into a new, independent vector.

Example

```
vec1i w = {1,2,3};
vec1u id = {0,1};
vec1i z(w[id]); // {1,2}

// Assignment
z = w[id];
```

2.2.3 Member functions

- **bool vec_t::empty() const**

This function will return **true** if this vector contains at least one element, and **false**

otherwise. In particular, `true` will be returned for default constructed vectors, and after a call to `vec_t::clear()`.

Example

```
vec1i v;  
v.empty(); // true  
vec1i w = {1,2,3};  
w.empty(); // false
```

- `std::size_t vec_t::size() const`

This function will return the total number of elements in this vector. If the vector is empty, then the function returns `0`. If the vector is multidimensional, then the function returns the product of all the dimensions.

Example

```
vec1i v;  
v.size(); // 0  
vec1i w = {1,2,3};  
w.size(); // 3  
vec2i z = {{1,2,3}, {4,5,6}};  
z.size(); // 6
```

- `void vec_t::resize(...)`

This function can be used to explicitly change the size of a vector. The parameters it accepts are the same as the dimension constructor (2.2.2), i.e., either integral values for individual dimensions, or an `std::array` containing multiple dimensions, or any combination of these. However, the total number of dimensions of the vector must remain unchanged.

After the vector has been resized, its content will have changed. For a monodimensional vector, if the resize operation *decreased* the total number of elements, then the last elements will be erased, but the other ones will remain untouched. On the other hand, if the resize operation *increased* the total number of elements, all the previous elements are unchanged, and new elements are inserted at the end of the vector, default constructed (i.e., zeroes for integral types, etc.). For a multidimensional vector, its content is left in an *undefined state*, and can be assumed to be destroyed⁴.

⁴This is just out of laziness. In the future, this will probably be specified to behave like the monodi-

Example

```
vec1i v = {1,2,3};
v.resize(5); // {1,2,3,0,0}
v.resize(2); // {1,2}
v.resize(3); // {1,2,0}

vec2f w = {{1,2},{2,3}};
w.resize(2,3); // w has been resized, but its content is unspecified
w(0,0); // ?
```

- **void** **vec_t::clear()**

This function removes all elements from the vector, and sets all dimensions to zero.

Example

```
vec1i v = {1,2,3};
v.clear(); // v is now empty
v.empty(); // true
v.size(); // 0
v.dims; // {0}
```

- T& **vec_t::back()**

This function is only available for monodimensional vectors. It returns the last element of the vector. Will crash if called on an empty vector.

Example

```
vec1i v = {1,2,3};
v.back(); // 3
v.back() == v[v.size()-1]; // always true
```

- T& **vec_t::front()**

This function is only available for monodimensional vectors. It returns the first element of the vector. Will crash if called on an empty vector.

mensional vector. The issue is that it involves non-trivial reshuffling of the elements to preserve the original structure after the dimensions have changed. I have not needed this feature so far.

Example

```
vec1i v = {1,2,3};  
v.front(); // 1  
v.front() == v[0]; // always true
```

- **void vec_t::push_back(...)**

The behavior of this function is different for monodimensional and multidimensional vectors. For monodimensional vectors, this function appends a new element at the end of the vector, and therefore takes for argument a single value of type T (or convertible to T). For multidimensional vectors, this function takes for argument another vector of D-1 dimensions, and whose lengths match the *last* D-1 dimensions of the first vector. This new vector is inserted after the existing elements, and the first dimension of the first vector is increased by one.

Example

```
vec1i v = {1,2,3};  
v.push_back(4); // {1,2,3,4}  
  
vec2i w = {{1,2,3}, {4,5,6}};  
w.push_back({7,8,9}); // {{1,2,3}, {4,5,6}, {7,8,9}}  
w.push_back({7,8}); // error: dimensions do not match, 2 != 3
```

- **void vec_t::reserve(std::size_t)**

This function is used for optimization, and is similar to `std::vector::reserve()` (actually, this function is called internally). To understand what this function actually does, one needs to know the internal behavior of `std::vector`. By default, the `std::vector` only allocates enough memory to hold a few elements. Once the allocated memory is full, `std::vector` allocates a larger amount of memory, copies the existing elements inside this new memory, and frees the old memory. This strategy allows virtually unlimited growth of a given vector, and is quite efficiently tuned. However, it is still an expensive operation, and performances can be greatly improved if one knows *in advance* the total number of objects that need to be stored in the vector, so that the right amount of memory is allocated from the beginning, and no copy is required. This function does just that, it tells `std::vector` how many elements it *will* (or might) contain at some point, so that the vector can already allocate enough memory. This is also useful if you only have a rough idea of the future number of elements.

Example

```
vec1i v = {1,2,3,...};

// Let's imagine you have an algorithm that will produce an
// unknown number of values, but you know that on average
// it is close to N*N.
vec1i w;
// Reserve roughly enough memory in advance
w.reserve(v.size()*v.size());

// Now the algorithm will run close to the optimal memory
// efficiency
for (...) {
    w.push_back(...);
}
```

- **uint_t vec_t::pitch(uint_t) const**

This is only useful for multidimensional vectors. This function returns the “pitch” factor associated to a given dimension. This factor is number of elements in memory that separate two consecutive indices of this dimension. By definition, the pitch factor is 1 for the last dimension. For the other dimensions, this is the product of all the other dimensions located between the one considered and the last dimension.

Example

```
vec3f v(5,8,6);
v.pitch(2); // 1
v.pitch(1); // 6 = v.dims[2]
v.pitch(0); // 48 = v.dims[2]*v.dims[1]
```

- **bool vec_t::is_same(const vec_t<D,T>&) const**

This function tests if the provided vector is a view inside this vector.

Example

```
vec1f v = {1,2,3};
vec1f w = {1,2,3};
vec1u id = {1,2};
```

```
v.is_same(v[id]); // true
v.is_same(w[id]); // false
```

- **const** **vec_t**& **vec_t::concretize()** **const**

This function just returns a reference to this vector, and is only present to mirror the interface of the view class.

- iterator **vec_t::begin()** and iterator **vec_t::end()**

These functions allow iteration over the values of this vector. The only reason one may use these functions explicitly is when using algorithms from the standard C++ library, which often work on a pair of iterators as returned by `begin()` and `end()`.

Example

```
vec1f v = {1,2,3};
// The presence of these functions allow to use vectors
// in range-based loops
for (float& f : v) {
    f += 1;
}
```

2.2.4 Indexing

There are two ways to index a given vector, for both mono- and multidimensional vectors. The first way is through “flat” indices and the bracket indexing `v[i]`, i.e., a single index that runs contiguously in memory, and the second way is through multidimensional indices and parenthesis indexing `v(i)` (2.1.2). For monodimensional vectors, these two methods are perfectly identical.

Flat indexing does not care about the details of the dimensions of a given vector. The only important thing is the total number of elements in the vector. This is the simplest and fastest⁵ way to access the data inside a vector.

Example

```
vec3f v(4,5,8); // a complex 3D vector
for (std::size_t i = 0; i < v.size(); ++i) {
```

⁵Actually there is a faster way using the `safe` wrapper, which does not do bounds checking. See the “Advanced” note at the end of this section.

```

    // Here we traverse the vector v regardless of its dimensions
    v[i] = 12.0 + i*i - sqrt(5.0*i) + v[i/4];
}

```

Multidimensional indexing is more involved computationally, because it implies some index arithmetic to compute the flat index and find the right place to read in memory. However it is much more expressive and easier to read and understand. Furthermore, except for very critical code sections, the performance gap is usually negligible, as long as you iterate over the dimensions properly, i.e., following the example below, that you do the *last* nested loop to iterate on the *last* index. This will guarantee as much memory locality as possible, and will take best advantage of CPU caches.

Example

```

vec3f v(4,5,8); // a complex 3D vector
for (std::size_t i = 0; i < v.dims[0]; ++i)
for (std::size_t j = 0; j < v.dims[1]; ++j)
for (std::size_t k = 0; k < v.dims[2]; ++k) {
    // Here we traverse the vector v keeping its dimensional structure
    v(i,j,k) = 42.0 + i + j + k/(j+1);
}

```

Indexing a vector can only be done with integers, e.g., **int**, **unsigned int**, **int_t**, **uint_t**, **std::size_t**, etc. Indexing with *unsigned* integers is faster because it removes the need to check for the positivity of the index, and should therefore be preferred when possible. Negative indices are allowed though, and they are interpreted as *reverse* indices, where -1 refers to the last element of the vector, -2 the one before the last, etc.

Example

```

vec1i v = {1,2,3,4};
v[0];    // 1, int index
v[0u];   // 1, unsigned int index (faster, but more cumbersome)
vec[-1]; // 4, int index
vec[-2]; // 3, int index
vec[-1] == vec[vec.size()-1]; // always true

v[0.1]; // error: can only access using integers

```

As discussed in the overview (2.1.3), one can also use vector to index another one.

This is called array indexing, by opposition to the scalar indexing we have seen just above. Similarly to scalar indexing, array indexing is only allowed by using integers, i.e., `vec1u`, `vec1i` and their multidimensional counterparts. Again, unsigned integers should be preferred when possible.

Example

```
vec1i v = {1,2,3,4};
vec1u id = {0,2,3};
v[id]; // 1,3,4
```

Sometimes, one will want to use array indexing to access all the elements at once, for example to set all the elements of a vector to a specific value. This can be done with a loop, of course, but it can be boring to write. To do so without explicitly writing the loop, one typically has to create first an index vector containing all the indices of the target vector, and then apply the operation:

```
vec1i v = {1,2,3,4};
vec1u id = {0,1,2,3}; // all the indices of v
v[id] = 12;

// Note that the following will not work
v = 12; // now v only contains a single element equal to 12
```

Not only is this not very practical to write, it is error prone and not very clear. If we decide to add an element to `v`, we also have to modify `id`. Not only this, but it will most likely be slower than writing the loop directly, because the compiler may not realize that you are accessing all the elements contiguously, and will fail to optimize it properly. For this reason, we also introduce the “placeholder” symbol, defined as a single underscore `_`. When used as an index, it means “all the indices in the range”. Coming back to our example:

```
vec1i v = {1,2,3,4};
v[_] = 12; // it cannot get much shorter
```

This placeholder index can be used in all situations, with both flat and multidimensional indexing. It can be further refined to only encompass a fraction of the whole range, using a peculiar syntax⁶:

⁶Be warned that this feature has been introduced recently and may not survive in the future.

```
vec1i v = {1,2,3,4};  
v[_-2] = 12;    // only access the indices 0 to 2  
v[2-_] = 12;    // only access the indices 2 to 3  
v[1--2] = 12;   // only access the indices 1 to 2
```

Except for the special case of the placeholder index `_`, all the indexing methods described above perform *bound checks* before accessing each element. In other words, the vector class makes sure that each index is smaller than either the total size of the vector (for flat indices) or the length of its corresponding dimension (for multidimensional indices). If this condition is not satisfied, an assertion is raised explaining the problem, and the program is stopped immediately to prevent memory corruption.

Advanced

This bound checking has a small but noticeable impact on performances. In most cases, the added security is definitely worth it. Indeed, accessing a vector with an out of bounds index has very unpredictable impacts on the behavior of the program: sometimes it will crash, but most of the time it will not. Memory will be silently corrupted, the problem will be hard to notice, but the consequences can be terrible... Then, identifying the root of the problem to fix it may prove even more challenging. This is why bound checking is enabled by default.

However, there are cases where bound checking is superfluous, for example if we already know *by construction* that the indices we are dealing with will always be valid. Sometimes the compiler may realize that and optimize the checks away, but one should not rely on it. If these situations are computation-limited, i.e., a lot of time is spent doing some number crushing for each element, then the performance hit of bound checking will be negligible, and one should not worry about it. On the other hand, if very little work is done per element, then most of the time will be spent iterating from one index to the next and loading the value in the CPU cache, and bound checking can take a significant amount of the total time.

For this reason, the `phy++` vector also offers an alternative indexing interface, the *safe* interface, that behaves exactly like the standard interface described above, except that it does not perform bound checking. One may access it using `v.safe[i]` for flat indexing, or `v.safe(x,y)` for multidimensional indexing, and it can also be used to create views. This interface is not meant to be used in daily coding, but rather for computationally intensive functions that you write once but use many times.

2.2.5 Operators

The set of available operators depend on the type of the elements contained in the vector.

- For arithmetic types (**int**, **unsigned int**, **float** and **double**): addition (+), subtraction (-), multiplication (*), division (/) and (integer types only) modulo (%).
- For **bool**: and (&&), or (||) and negation (!).
- For `std::string`: concatenation (+).
- For all types: less than (<), less than or equal (<=), greater than (>), greather than or equal (>=), equal (==) and not equal (!=).

In all cases, the operator overloading allows mixing together vectors of different types (as long as they are convertible one to another) and scalar values.

Example

```
vec<int> v = {1,2,3,4};  
v + 2; // {3,4,5,6}  
v + v; // {2,4,6,8}
```

```
vec<string> fruits = {"apple", "orange"};  
"I ate an " + fruits; // {"I ate an apple", "I ate an orange"}  
2 + fruits;           // error: no operator found for int + std::string
```

2.3 The view class

The full type of the view class is

```
template<std::size_t Dimension, typename ElementType>  
struct vec_t<Dimension, ElementType*>;  
//           note the asterisk ^
```

The public interface of the view class is very similar to that of the normal vector, and we will not repeat it here. There are some important differences though, which are inherent to the goal of this class. In particular, there is no available constructor (you do now create a view yourself, you ask for it from an existing vector that will create it for you), and the `resize()` function is not available. Lastly, the view implements `concretize()`, differently.

Thanks to their strong similarity, we will not distinguish in the other sections between vectors and views, and will consider views as just another kind of vectors. Indeed, the interface of these two classes has been designed for views to be completely interchangeable with vectors, and vice versa, so that the code of any given function is generally written once and is valid for both types.

2.3.1 Member functions

- `vec_t<D,T> vec_t<D,T*>::concretize() const`

This function creates a new vector out of the elements of this view. The returned vector is completely independent from this view, or the original vector this view is currently pointing to. This function is mostly useful when writing generic functions. Indeed, views are implicitly convertible to normal vectors on assignment (2.2.2).

2.4 Metaprogramming traits and functions (WIP)

Warning

This whole section is more advanced than the rest. It is describing the sets of helper types and metaprogramming functions that one can use to write new functions. It is intended to be followed by readers already familiar with the `phy++` library, and with good knowledge of C++ metaprogramming.

Chapter 3

Support libraries

In this chapter we describe the set of helper functions that are part of the `phy++` support library. These functions are not essential to the use of the `phy++` library, but are mostly modular components that one may choose to use or not. All the support functions are sorted into broad categories to help you discover new functions and algorithm. Alternatively, if you know the name of a function and would like to read its documentation, an index is available at the end of this document.

Note that, in all this section, the signature of the functions is given in pseudo-code, both for conciseness and readability. In particular, the following rules apply.

- The presence of the \textcircled{V} symbol in front of the signature of the function means that this function is also available in a *vectorized* form. This only applies to functions whose first argument is a scalar (i.e., not a vector). In this case, the vectorized form shares the same signature as the original form, but the first argument is promoted to a vector. Calling the vectorized version is the same as writing a loop to call the original version on each element of the vector. If the original function had a return value, the vectorized form returns a vector whose elements are the return value of each call, corresponding to each element of the input vector. The vectorized version can be faster than writing the loop manually.

Example

```
// Suppose this function is marked as vectorized
bool is_odd(uint_t)
// It means that there is another function with the
// same name, but that acts on a vector instead
vec_t<D, bool> is_odd(vec_t<D, uint_t>)
```

```
// It is used like this
vec1u v = {1,2,3,4,5};
vec1b b = is_odd(v);
// ... and is equivalent to
vec1b b(v.dims);
for (uint_t i : range(v)) {
    b[i] = is_odd(v[i]);
}
```

- If the function depends on an external library, the name of this library will be written before the signature, for example if LAPACK is needed you will find the symbol [LAPACK].
- Template parameters are not declared explicitly. They are always written in upper-case, usually with a single character (e.g., T, D), or possibly two (e.g., a letter and a number), but never more. Letters T and U refer to template *types*, while letters D, N or I refer to template *integers*.

Example

```
// Pseudo-code used in this section
void foo(T)
// Corresponding C++ code
template<typename T>
void foo(T);

// Pseudo-code used in this section
void foo(vec_t<D,T> v, U u)
// Corresponding C++ code
template<std::size_t D, typename T, typename U>
void foo(vec_t<D,T> v, U u);
```

- Template parameters are omitted when not relevant to the description of the function. In this case, it is implicitly assumed that the function will work for any type/value of these template parameters.

Example

```
// Pseudo-code used in this section
```

```

void sort(vec_t&)
// Corresponding C++ code
template<std::size_t D, typename T>
void sort(vec_t<D,T>&);

```

- There are only two kinds of arguments: input arguments, and input/output arguments. Input arguments are always spelled as plain types, e.g. T, even if the actual signature of the function uses a constant reference, an r-value reference or a universal reference. The reason is that this implementation choice does not matter to the end user. What matters is the interface. The input/output parameters are always C++ references, e.g. T&.

Example

```

// Pseudo-code used in this section
vec1u dims(vec_t)
// Corresponding C++ code could be either
template<std::size_t D, typename T>
vec1u dims(vec_t<D,T>);
// ... or
template<std::size_t D, typename T>
vec1u dims(const vec_t<D,T>&);
// The only difference is that the first version will
// always make a copy (or move) of its parameter, while
// the second may not. This optimization choice depends
// on the actual code inside the function, and has no
// consequence on how the function is actually used.

```

- The ellipsis ... is used to symbolize a list of multiple arguments whose length can vary depending on the context. These arguments are not spelled out explicitly, but the description of the function must make it clear what they are used for. Optionally, a type may be placed before the ellipsis to indicate that all the arguments must be of this same type.

Example

```

// Pseudo-code used in this section
uint_t flat_id(vec_t, ...)
// Corresponding C++ code

```

```
template<std::size_t D, typename T, typename ... Args>
uint_t flat_id(vec_t<D,T>, Args&& ...);
```

- The std namespace is omitted for common standard types, like std::string or std::array.

Example

```
// Pseudo-code used in this section
uint_t length(string)
// Corresponding C++ code
uint_t length(const std::string&);
```

- For the particular case of std::array, the individual elements inside the array can be named by placing the names inside curly braces after the type of the array.

Example

```
// Pseudo-code used in this section
uint_t distance(array {x,y})
// Corresponding C++ code
template<typename T>
uint_t length(const std::array<T,2>& a) {
    // x := a[0]
    // y := a[1]
}
```

- If the return type of a function in pseudo-code is **auto**, it means that this return value is “complex” (usually a structure or a class) and it is not important to know its precise type. The description of the function must therefore make it clear how this return value can be used.

With this in mind, it is clear that this chapter focuses on the *interface* that is provided by the library, rather than on the individual function themselves. In fact, a single interface may be composed of many different functions to take care of all the combination of types that the interface supports. If the reader is interested in all these overloads, or is experiencing a particular compiler error that cannot be easily fixed just by looking at the interface, then it is best to look directly into the code of the library. Although less readable than the pseudo-code used in this document, most of the time an effort is made to make the code as clear as possible. However, if a function is too hard to understand, I consider this as a bug that

should be reported on the `github` issue tracker (seriously). Similarly, if you end up doing something wrong with the library, but that the compiler error message is too cryptic or too long, you may also fill in a bug report. Ensuring that clear error messages are sent to the user is a shared responsibility between compiler writers and library authors.

3.1 Generic vector functions

The vector and view classes are useful and complete tools. However, there are a number of tasks that one repeatedly need to do, like generating a sequence of indices, or sorting a vector, and that would be tedious to write each time they are needed. For this reason, the `phy++` library comes with a large set of utility functions to sort, rearrange, and select data inside vectors. In this section we list these functions and algorithms.

This support library also introduces a global constant called `npos`. This is an unsigned integer whose value is the largest possible integer that the `uint_t` type can hold. It is meant to be used as an error value, or the value to return if no normal value would make sense. It is very similar in concept to the `std::string::npos` provided by the C++ standard library. In particular, it is worth noting that converting `npos` to a *signed* integer produces the value `-1`.

We now describe the functions provided by this library, sorted by categories.

3.1.1 Range-based iteration

- `auto range(vec_t v)`
`auto range(uint_t n)`
`auto range(uint_t i0, uint_t n)`

This function returns a C++ *range*, i.e., an object that can be used inside the C++ range-based `for` loop. This range will generate integer values starting from `0` (first and second version) or `i0` (third version) to `v.size()` (first version) or `n` (second and third versions), that last value being *excluded* from the range. This nice way of writing an integer `for` loop actually runs as fast as (if not faster than) the classical way, and is less error prone.

Example

```
vec1i v = {4,5,6,8};
```

```
// First version
```

```

for (uint_t i : range(v)) {
    // 'i' goes from 0 to 3
    v[i] = ...;
}

// Note that the loop above generates
// *indices* inside the vector, while:
for (int i : v) { /* ... */ }
// ... generates *values* from the vector.

// Second version
for (uint_t i : range(3)) {
    // 'i' goes from 0 to 2
    v[i] = ...;
}

// Third version
for (uint_t i : range(1,3)) {
    // 'i' goes from 1 to 3
    v[i] = ...;
}

```

3.1.2 Index manipulation

- `vec1u mult_ids(vec_t v, uint_t i)`

This is only useful for multidimensional vectors. This function converts a “flat” index `i` ([2.2.4](#)) into an array of multidimensional indices, following the dimensions of the provided vector `v`. The `flat_id` function does the inverse job.

Example

```

vec2i v(2,3);
mult_ids(v,0); // {0,0}
mult_ids(v,1); // {0,1}
mult_ids(v,2); // {0,3}
mult_ids(v,3); // {1,0}
v[3] == v(1,0); // true

```

- `uint_t flat_id(vec_t v, ...)`

This is only useful for multidimensional vectors. This function converts a group of

multidimensional indices into a “flat” index (2.2.4), following the dimensions of the provided vector `v`. The `mult_ids` function does the inverse job.

Example

```
vec2i v(2,3);
flat_id(v,0,0); // 0
flat_id(v,0,1); // 1
flat_id(v,0,2); // 2
flat_id(v,1,0); // 3
v(1,0) == v[3]; // true
```

3.1.3 Integer sequences

- `vec_t<D,int_t> indgen(...)`
`vec_t<D,uint_t> uindgen(...)`
`vec_t<D,float> findgen(...)`
`vec_t<D,double> dindgen(...)`

These functions will create a new vector, whose dimensions are specified in argument (like in the dimension constructor, 2.2.2, or `vec_t::resize()`, 2.2.3). After this vector is created, the function will fill it with values that start at 0 and increment by steps of 1 until the end of the vector.

Example

```
vec1i v = indgen(5); // {0,1,2,3,4}
vec2u w = uindgen(3,2); // {{0,1}, {2,3}, {4,5}}
```

3.1.4 Rearranging elements

- `vec1u sort(vec_t)`
`void inplace_sort(vec_t&)`

These functions will change the order of the elements inside a given vector so that they are sorted from the smallest to the largest. The difference between the two versions is that `sort` does not actually modify the provided vector, but rather returns a vector containing indices inside the provided vector, and `inplace_sort` directly modifies the provided vector. The later is the fastest of the two, but it is less powerful.

Example

```
// First version
vec<li> v = {1,5,6,3,7};
vec<li> id = sort(v); // {0,3,1,2,4}
v[id]; // {1,3,5,6,7} is sorted
// now, id can also be used to modify the order of
// another vector of the same dimensions

// Second version
inplace_sort(v);
v; // {1,3,5,6,7} is sorted
```

- **bool** is_sorted(**vec_t**)

This function just traverses the whole input vector and checks if its elements are sorted by increasing value.

Example

```
// First version
vec<li> v = {1,5,6,3,7};
is_sorted(v); // false
inplace_sort(v);
v; // {1,3,5,6,7}
is_sorted(v); // true
```

- **vec_t<1,T>** reverse(**vec_t<1,T>**)

This function will inverse the order of all the elements inside the provided vector.

Example

```
vec<li> v = {1,2,3,4,5,6};
vec<li> w = reverse(v); // {6,5,4,3,2,1}
```

- **vec_t<1,T>** shift(**vec_t<1,T>** v, **int_t** n, T d = 0)

This function will shift the position of the elements inside the provided vector v by a given amount of indices n. Elements that would go outside of the bounds of the vector are destroyed. New elements are inserted and default constructed, or assigned the default value d (optional argument).

Example

```
vec1i v = {1,2,3,4,5};
vec1i sr = shift(v, 2);
sr; // {0,0,1,2,3}
vec1i sl = shift(v, -2, 99);
sl; // {3,4,5,99,99};
```

3.1.5 Finding elements

- `vec1u where(vec_t<D, bool>)`

This function will scan the **bool** vector provided in argument, will store the flat indices (2.2.4) of each element which is **true**, and will return all these indices in a vector. This is a very useful tool to filter and selectively modify vectors, and probably one of the most used function of the whole library.

Example

```
vec1i v = {4,8,6,7,5,2,3,9,0};
// We want to select all the elements which are greater than 3
// We use where() to get their indices
vec1u id = where(v > 3); // {0,1,2,3,4,7}
// Now we can check
v[id]; // {4,8,6,7,5,9}, good!

// The argument of where() can be as complex as you want
id = where(v < 3 || (v > 3 && v % 6 < 2)); // now guess

// It can also involve multiple vectors, as long as they have
// the same dimensions
vec1i w = {9,8,6,1,-2,0,8,5,1};
id = where(v > w || (v + w) % 5 == 0);
// The returned indices are valid for both v and w
v[id]; // {8,6,7,5,2,9}
w[id]; // {8,6,1,-2,0,5}
```

- `vec1u complement(vec_t v, vec1u id)`

This function works in tandem with `where`. Given a vector `v` and a set of flat indices `id` (2.2.4), it will return the complementary set of indices inside this vector, i.e., all the indices of `v` that are *not* present in `id`.

Example

```
vec1i v = {1,5,6,3,7};  
vec1u id = where(v > 4); // {1,2,4}  
vec1u cid = complement(v, id); // {0,3}
```

- **uint_t** lower_bound(T x, **vec_t** v)
uint_t upper_bound(T x, **vec_t** v)
array<**uint_t**,2> bounds(T x, **vec_t** v)
array<**uint_t**,2> bounds(T x1, U x2, **vec_t** v)

These functions use a binary search algorithm to locate the element in the input vector *v* that is equal to or closest to the provided value *x*. It is important to note that the binary search assumes that the elements in the input vector are *sorted* by increasing value. This algorithm also assumes that the input vector does not contain any NaN value (3.8).

The lower_bound function locates the last element in *v* that is less or equal to *x*. If no such element is found, *npos* is returned.

The upper_bound function locates the first element in *v* that is greater than *x*. If no such element is found, *npos* is returned.

The first bounds function combines what both lower_bound and upper_bound do, and returns both indices in an array. The second bounds function calls lower_bound to look for *x1*, and upper_bound to look for *x2*.

Example

```
vec1i v = {2,5,9,12,50};  
bounds(0, v); // {npos,0}  
bounds(9, v); // {2,3}  
bounds(100, v); // {4,npos}
```

- vec1u equal_range(T x, **vec_t** v)

Similarly to lower_bound, upper_bound and bounds, this function uses a binary search algorithm to locate all the values in the input vector *v* that are equal to *x*. It then returns the flat indices of all these values in a vector. It is important to note that the binary search assumes that the elements in the input vector are *sorted* by increasing value. This algorithm also assumes that the input vector does not contain any NaN value (3.8).

If no such value is found, an empty vector is returned.

Example

```
vec1i v = {2,2,5,9,9,9,12,50};  
equal_range(9, v); // {3,4,5}
```

```
// It's a faster version of  
where(v == 9);
```

- `vec1u uniq(vec_t v)`


`vec1u uniq(vec_t v, vec1u sid)`

This function will traverse the provided vector `v` and find all the unique values. It will store the indices of these values (if a value is present more than once inside `v`, the index of the first one will be used) and return them inside an index vector. The first version assumes that the values in `v` are *sorted* from the smallest to the largest. In the second version, `v` may not be sorted, but the second argument `id` contains indices that will sort `v` (e.g., `id` can be the return value of `sort(v)`).

Example

```
// For a sorted vector  
vec1i v = {1,1,2,5,5,6,9,9,10};  
vec1u u = uniq(v); // {0,2,3,5,6,8}  
v[u]; // {1,2,5,6,9,10} only unique values
```

```
// For an non-sorted vector  
vec1i w = {5,6,7,8,6,5,4,1,2,5};  
vec1u u = uniq(w, sort(w));  
w[u]; // {1,2,4,5,6,7,8}
```

-  `bool is_any_of(T1 v1, vec_t<D2,T2> v2)`

This function looks if there is any value inside `v2` that is equal to `v1`. If so, it returns `true`, else it returns `false`.

Example

```
vec1i v = {7,4,2,1,6};
```

```
vec1i d = {5,6,7};  
vec1b b = is_any_of(v, d); // {true, false, false, false, true}
```

- **void** match(**vec_t** v1, **vec_t** v2, vec1u& id1, vec1u& id2)

This function traverses v1 and, for each value in v1, looks for elements in v2 that have the same value. If one is found, the flat index of the element of v1 is added to id1, and the flat index of the element of v2 is added to id2. Then the function goes on to the next value in v1. Note that, contrary to match_dictionary, each value in v1 is matched to *at most* one value in v2, and vice versa. In other words, if v1 or v2 contain duplicates, only the first value will be matched and the others will be ignored. This function is symmetric: the result will be the same if you swap the two input vectors (of course, the output vectors have to be swapped also).

Example

```
vec1i v = {7,4,2,1,6};  
vec1i w = {2,6,5,3};  
vec1u id1, id2;  
match(v, w, id1, id2);  
id1; // {2,4}  
id2; // {0,1}  
v[id1] == w[id2]; // always true
```

- **void** match_dictionary(**vec_t** v1, **vec_t** v2, vec1u& id1, vec1u& id2)

This function traverses v1 and, for each value in v1, looks for an element in v2 that has the same value. If one is found, the flat index of the element of v1 is added to id1, and the flat index of the element of v2 is added to id2. Then the function goes on to the next value in v1. Contrary to match, each value in v2 can be matched to multiple values in v1. Therefore this function is not symmetric: the second vector should be considered as a “dictionary”, hence the name of this function. It is assumed that v2 does not contain any duplicates.

Example

```
vec1i v = {7,6,2,1,6};  
vec1i w = {2,6,5,3};  
vec1u id1, id2;  
match_dictionary(v, w, id1, id2);
```

```
id1; // {1,2,4}
id2; // {1,0,1}
v[id1] == w[id2]; // always true
```

- **bool** astar_find(vec2b m, **uint_t**& x, y)

This function uses the A^* (“A star”) algorithm to look inside a 2D boolean map **m** and, starting from the position **x** and **y** (i.e. **m**(**x**,**y**)), find the closest point that has a value of **true**. Once this position is found, its indices are stored inside **x** and **y**, and the function returns **true**. If no element inside **m** is **true**, then the function returns **false**.

Example

```
// Using '@' for true and '.' for false,
// assume we have the following boolean map,
// and that we start at the position indicated
// by 'S', the closest point whose coordinates
// will be returned by astar_find is indicated
// by an 'X'
vec2b m;
// .....
// .....
// .....
// ...aaaaa.....
// ...aaaaa.....
// ...aaaaa.....
// ...aaaaa.....
// ...aaaaa.....
// ...aaaaaX.....
// .....S....
// .....
// .....

uint_t x = 13, y = 2;
astar_find(m, x, y);
x; // 7
y; // 3
```

3.1.6 Modifying dimensions

- **vec_t**<1,T> flatten(**vec_t**<D,T>)

This function transforms a multidimensional vector into a monodimensional vector. The content in memory is exactly the same, so the operation is fast. In particular, if the argument of this function is a temporary, this function is extremely cheap as it produces no copy. The `reform` function does the inverse job, and more.

Example

```
vec2i v = {{1,2,3}, {4,5,6}};
vec1i w = flatten(v); // {1,2,3,4,5,6}
```

- **vec_t**<D1,T> `reform`(**vec_t**<D2,T>, ...)

This function transforms a vector into another vector just by changing its dimensions. The content in memory is exactly the same, so the operation is fast. In particular, if the argument of this function is a temporary, this function is extremely cheap as it produces no copy. However, the new dimensions have to generate the same number of elements as the provided vector contains. The `flatten` function is a special case of `reform`.

Example

```
vec1i v = {1,2,3,4,5,6};
vec2i w = reform(v, 2, 3); // {{1,2,3}, {4,5,6}}
```

- **vec_t**<N,T> `replicate`(T, ...)
- **vec_t**<D+N,T> `replicate`(**vec_t**<D,T>, ...)

This function will take the provided scalar (first version) or vector (second version), and replicate it multiple times according to the provided additional parameters, to generate additional dimensions.

Example

```
// First version
vec1i v = replicate(2, 5); // {2,2,2,2,2} 5 times 2
vec2i w = replicate(2, 3, 2); // {{2,2},{2,2},{2,2}} 3 x 2 times 2

// Second version
vec2i z = replicate(vec1i{1,2}, 3); // {{1,2},{1,2},{1,2}} 3 times {1,2}
// Note that it is not possible to just use a plain initializer list
```

```
vec2i z = replicate({1,2}, 3); // error, unfortunately
```

3.1.7 Adding/removing elements

- **vec_t**<1,T> remove(**vec_t**<1,T> v, vec1u id)
void inplace_remove(**vec_t**<1,T>& v, vec1u id)

These functions will remove the elements in **v** that have their indices in **id**. The only difference between the first and the second version is that the former will first make a copy of the provided vector, remove elements inside this copy, and return it. The second version modifies directly the provided vector, and is therefore faster.

Example

```
// First version
vec1i v = {4,5,2,8,1};
vec1i w = remove(v, {1,3}); // {4,2,1}

// Second version
inplace_remove(v, {1,3});
v; // {4,2,1}
```

- **void** append<N>(**vec_t**<D,T1>& t1, **vec_t**<D,T2> t2)
void append(**vec_t**<1,T1>& t1, **vec_t**<1,T2> t2)
void prepend<N>(**vec_t**<D,T1>& t1, **vec_t**<D,T2> t2)
void prepend(**vec_t**<1,T1>& t1, **vec_t**<1,T2> t2)

These functions behave similarly to **vec_t::push_back**, in that they will add new elements at the end (append), but also at the beginning (prepend) of the provided vector. However, when **vec_t::push_back** can only add new elements from a vector that is one dimension less than the original vector (or a scalar, for monodimensional vectors), these functions will add new elements from a vector of the *same* dimension. These functions are also more powerful than **vec_t::push_back**, because they allow you to choose along which dimension the new elements will be added, through the template parameter **N** (note that this parameter is useless and therefore does not exist for monodimensional vectors). The other dimensions must be identical. They are clearly dedicated to a more advanced usage.

Example

```
// For monodimensional vectors
vec1i v = {1,2,3};
vec1i w = {4,5,6};
append(v, w);
v; // {1,2,3,4,5,6}
prepend(v, w);
v; // {4,5,6,1,2,3,4,5,6}

// For multidimensional vectors
vec2i x = {{1,2}, {3,4}}; // x is (2x2)
vec2i y = {{0}, {0}}; // y is (1x2)
vec2i z = {{5,6,7}}; // z is (3x1)
append<1>(x, y);
x; // {{1,2,0}, {3,4,0}} // x is (2x3)
prepend<0>(x, z);
x; // {{5,6,7}, {1,2,0}, {3,4,0}} // x is (3x3)
```

3.1.8 Error checking

- **void** phypp_check(**bool** b, ...)

This function makes error checking easier. When called, it checks the value of `b`. If `b` is `true`, then nothing happens. However if `b` is `false`, then the current file and line are printed to the standard output, followed by the other arguments of this function (they are supposed to be an error message explaining what went wrong), and an assertion is raised, immediately stopping the program. This function is used everywhere in the `phy++` library to make sure that certain conditions are properly satisfied before doing a calculation, and it is essential to make the program crash in case something is unexpected (rather than letting it run hoping for the best, and often getting the worst).

Since this function is actually implemented by a preprocessor macro, you should not worry about its performance impact. It will only affect the performances when something goes wrong and the program is about to crash. However the calculation of `b` can itself be costly (for example, you may want to check that a vector is sorted), and there is no way around this.

Example

```
vec1i v;
// Suppose v is read from the command line arguments.
v = read_from_somewhere();






// The following code needs at least 3 elements in the
// vector v, so we need to check that first.
phypp_check(v.size() >= 3, "this algorithm needs at least "
    "3 values in the input vector, but only ", v.size(),
    " were found");

// If we get past this point, then v has the right number
// of elements, and we can proceed
do_stuff(v);
```

3.2 Parsing command line arguments

- **void** read_args(**int** argc, **char*** argv[], ...)

3.3 File input/output

-  **bool** file::exists(string)
- **bool** file::is_older(string f1, string f2)
- vec1s file::list_directories(string)
- vec1s file::list_files(string)
-  string file::directorize(string)
-  string file::get_basename(string)
-  string file::get_directory(string)
-  **bool** file::mkdir(string)
- **void** file::read_table(string f, **uint_t** s, ...)

- `void file::write_table(string f, uint_t w, ...)`
`void file::write_table_csv(string f, uint_t w, ...)`
`void file::write_table_hdr(string f, uint_t w, vec1s hdr, ...)`

3.4 String manipulation

- `string strn(T)`
`string strn(T v, uint_t n, char fill = '0')`
`string strn_sci(T v)`
- `vec_<D,string> strna(vec_t<D,T>)`
`vec_<D,string> strna(vec_t<D,T> v, uint_t n, char fill = '0')`
`vec_<D,string> strna_sci(vec_t<D,T> v)`
- \textcircled{V} `bool from_string(string s, T& v)`
- \textcircled{V} `bool empty(string s)`
- \textcircled{V} `uint_t length(string s)`
- \textcircled{V} `string trim(string s, string cs)`
- \textcircled{V} `string toupper(string s)`
- \textcircled{V} `string tolower(string s)`
- \textcircled{V} `string replace(string s, string p, string r)`
- `vec1s split(string s, string p)`
- `vec1s cut(string s, uint_t n)`
- `vec1s wrap(string s, uint_t w, string i = "", bool e = false)`
- `string collapse(vec_t<D,string>)`
`string collapse(vec_t<D,string> v, string s)`
- \textcircled{V} `uint_t find(string s, string p)`

- \textcircled{V} **bool** match(string s, string r)
- \textcircled{V} **bool** match_any_of(string s, vec1s r)
- \textcircled{V} **bool** start_with(string s, string p)
- \textcircled{V} **bool** end_with(string s, string p)
- \textcircled{V} string erase_begin(string s, string p)
- \textcircled{V} string erase_begin(string s, **uint_t** n)
- \textcircled{V} string erase_end(string s, string p)
- \textcircled{V} string erase_end(string s, **uint_t** n)
- \textcircled{V} string keep_start(string s, **uint_t** n = 1)
- \textcircled{V} string keep_end(string s, **uint_t** n = 1)
- \textcircled{V} string remove_extension(string)
- \textcircled{V} string align_left(string s, **uint_t** w, **char** f = ' ')
- \textcircled{V} string align_center(string s, **uint_t** w, **char** f = ' ')
- \textcircled{V} string align_right(string s, **uint_t** w, **char** f = ' ')
- \textcircled{V} **uint_t** distance(string s1, string s2)

3.5 OS interaction

- string system_var(string v, string d = "")

3.6 Printing to the terminal

- **void** print(...)
- **void** error(...)
- **void** warning(...)
- **void** note(...)

3.7 Measuring time

- `double` now()
- string today()
- string time_str(`double`)
- string seconds_str(`double`)
- `auto` progress_start(`uint_t`)
 `void` progress(`auto` p, `uint_t` m)
 `void` print_progress(`auto` p, T i, `uint_t` m)

3.8 Mathematics

The mathematics support library is among the largest inside `phy++`. It contains many functions, as well as a handful of useful constants. Some functionalities of this library are only available if you have installed the `fftw` and `gsl` libraries. If not, the specific functions that depend on these libraries will not be available, or will be slow, but the rest of the library will function properly.

This library provides the following global constants:

- `fnan` and `dnan`. These are the `float` and `double` representation of the “not-a-number” (NaN) special value. This value is returned by some operations that are mathematically undefined in the real domain. For example, dividing zero by zero, or taking the square root of a negative number. NaN has some very peculiar properties that can surprise the newcomer. In particular, it propagates extremely fast, since any operation involving at least a NaN value will always return NaN, e.g., `2.0 + fnan == fnan`). More troubling, any comparison operation involving a NaN will return `false`, e.g., `(10.0 < fnan) == false` and `(10.0 >= fnan) == false` too. The only notable exception to this rule is that `(fnan != fnan) == true`. Knowing this, NaN is a very useful return value to indicate that giving an actual value would not make sense. For example, in a galaxy catalog, some galaxies may have been observed at a certain wavelength, but not all of them. For those that are not observed, we do not know their flux. In this case, astronomers typically assign them a special, weird value, such as `-99`. Using NaN in this case is clearer.
- `fpi` and `dpi`. This is the `float` and `double` closest representation of the number $\pi = 3.14519\dots$

- `finf` and `dinf`. This is the **float** and **double** representation of the positive infinity. The positive infinity is larger than any other finite value.

We now present the functions provided by this support library. One of the responsibilities of this library is to bring vectorized versions of standard mathematical functions that only work for scalar values. Since these functions are fairly common and well known, we will not describe their signature and behavior, and instead just list them here:

- exponentiation: `sqrt`, `pow`,
- trigonometry: `cos`, `sin`, `tan`, `acos`, `asin`, `atan`, `cosh`, `sinh`, `tanh`, `acosh`, `asinh`, `atanh`,
- exponentials and logarithms: `exp`, `log`, `log2`, `log10`,
- special functions: `erf`, `erfc`, `tgamma`,
- rounding: `ceil`, `floor`, `round`,
- absolute value: `fabs`.

We also introduce the functions `bessel_j0`, `bessel_j1`, `bessel_y0`, `bessel_y1`, `bessel_i0`, `bessel_i1`, `bessel_k0`, `bessel_k1`. The scalar version of the first four are provided by the C++ standard, while the last four are provided by the `gsl`.

We now list the other, less common functions provided in this library. These are grouped by sections.

3.8.1 Low level mathematics

- \textcircled{V} **double** `e10(double)`
- \textcircled{V} `T` `sqr(T)`
- \textcircled{V} `T` `invsqr(T)`
- \textcircled{V} `T` `clamp(T t, U mi, V ma)`
- \textcircled{V} **bool** `finite(T)`
- \textcircled{V} **bool** `nan(T)`
- \textcircled{V} **int_t** `sign(T)`

3.8.2 Sequences and bins

- `vec1u rgen(T)`
`vec_t<1,T> rgen(T i, U j)`
`vec1d rgen(T i, U j, V n)`
`vec1d rgen_log(T i, U j, V n)`
- `vec_t<2,T> make_bins(T mi, T ma)`
`vec_t<2,T> make_bins(T mi, T ma, uint_t n)`
`vec_t<2,T> make_bins(vec_t<1,T>)`
- \textcircled{V} `bool in_bin(T v, vec_t<1,U> b)`
 \textcircled{V} `bool in_bin(T v, vec_t<2,U> b, uint_t ib)`
 \textcircled{V} `bool in_bin_open(T v, vec_t<2,U> b, uint_t ib)`
- `vec_t<1,T> bin_center(vec_t<2,T>) T bin_center(vec_t<1,T>)`
- `vec_t<1,T> bin_width(vec_t<2,T>) T bin_width(vec_t<1,T>)`

3.8.3 Randomization

- `auto make_seed(T)`
- `double randomn(auto seed)`
`vec_t<N,double> randomn(auto seed, ...)`
- `double randomu(auto seed)`
`vec_t<N,double> randomu(auto seed, ...)`
- `T randomi(auto seed, T mi, T ma)`
`vec_t<N,T> randomi(auto seed, T mi, T ma, ...)`
- `T random_pdf(auto seed, vec_t<1,T> x, vec_t<1,U> y, ...)`
- `bool random_coin(auto seed, double p)`
`vec_t<N,bool> random_coin(auto seed, double p, ...)`
- `vec_t shuffle(vec_t, auto seed)`
`void inplace_shuffle(vec_t&, auto seed)`

3.8.4 Reduction

- **void** run_dim_idx(**uint_t** d, F f, ...)
- **vec_t**<D-1,T> run_dim(**vec_t**<D,T> v, **uint_t** d, F f)
- T total(**vec_t**<D,T> v)
 vec_t<D-1,T> partial_total(**uint_t** d, **vec_t**<D,T> v)
- **uint_t** count(**vec_t**<D,**bool**> v)
 vec_t<D-1,**uint_t**> partial_count(**uint_t** d, **vec_t**<D,**bool**> v)
- **uint_t** fraction_of(**vec_t**<D,**bool**> v)
 vec_t<D-1,**uint_t**> partial_fraction_of(**uint_t** d, **vec_t**<D,**bool**> v)
- **double** mean(**vec_t** v)
 vec_t<D-1,**double**> partial_mean(**uint_t** d, **vec_t**<D,T> v)
- T median(**vec_t**<D,T> v)
 T inplace_median(**vec_t**<D,T>& v)
 vec_t<D-1,T> partial_median(**uint_t** d, **vec_t**<D,T> v)
- T percentile(**vec_t**<D,T> v, **double** p)
 vec_t<D-1,T> partial_percentile(**uint_t** d, **vec_t**<D,T> v, **double** p)
 vec_t<1,T> percentiles(**vec_t**<D,T> v, **double** ...)
- T min(**vec_t**<D,T> v)
 T min(**vec_t**<D,T> v, **uint_t**& i)
 vec_t<D-1,T> partial_min(**uint_t** d, **vec_t**<D,T> v)
 T max(**vec_t**<D,T> v)
 T max(**vec_t**<D,T> v, **uint_t**& i)
 vec_t<D-1,T> partial_max(**uint_t** d, **vec_t**<D,T> v)
- **uint_t** min_id(**vec_t**<D,T> v)
 uint_t max_id(**vec_t**<D,T> v)

- `vec1u histogram(vec_t v, vec_t<2,V> b)`
`vec_t<1,T2> histogram(vec_t<D,T1> v, vec_t<D,T2> w, vec_t<2,U> b)`
- `vec2u histogram2d(vec_t<D,T1> x, vec_t<D,T2> y, vec_t<2,U1> bx, vec_t<2,U2> by)`
- `vec_t<D,bool> sigma_clip(vec_t<D,T> v, double pl, double pu)`
- `vec_t<D,bool> mad_clip(vec_t<D,T> v, double s)`

3.8.5 Interpolation

- `double interpolate(double y1, y2, x1, x2, x)`
`T interpolate(vec_t<D,T> y, vec_t<D,U> x, V nx)`
`vec_t<D2,T> interpolate(vec_t<D1,T> y, vec_t<D1,U> x, vec_t<D2,V> nx)`
- `T bilinear(vec_t<2,T> m, double x, double y)`
- `vec_t<2,T> rebin(vec_t<2,T> m, vec1d mx, vec1d my, vec1d nx, vec1d ny)`

3.8.6 Calculus

- `double derivate1(F f, double x, double e)`
`double derivate1(F f, vec1d x, double e, uint_t i)`
- `double derivate2(F f, double x, double e)`
`double derivate2(F f, vec1d x, double e, uint_t i)`
- `double integrate_trap(F f, double x, uint_t n)`
`double integrate(F f, double x, double e = default)`
- `double integrate(vec_t<1,T> x, vec_t<1,U> y)`
`double integrate(vec_t<1,T> x, vec_t<1,U> y, double x0, double x1)`
`double integrate(vec_t<2,T> x, vec_t<1,U> y)`
`double integrate(vec_t<2,T> x, vec_t<1,U> y, double x0, double x1)`

3.8.7 Algebra

- `vec_t<2,V> mmul(vec_t<2,T> a, vec_t<2,U> b)`
`vec_t<1,V> mmul(vec_t<2,T> a, vec_t<1,U> b)`
`vec_t<1,V> mmul(vec_t<1,T> b, vec_t<2,U> a)`
- `vec_t<2,T> transpose(vec_t<2,T>)`
- `vec_t<1,T*> diagonal(vec_t<2,T>)`
- `vec2d identity_matrix(uint_t)`
- `vec2d scale_matrix(double sx, double sy)`
`vec2d scale_matrix(double s)`
- `vec2d translation_matrix(double dx, double dy)`
- `vec2d rotation_matrix(double a)`
- `vec1d point2d(double x, double y)`
- [LAPACK] `bool invert(vec2d a, vec2d& i)`
[LAPACK] `bool inplace_invert(vec2d&)`
- [LAPACK] `bool invert_symmetric(vec2d a, vec2d& i)`
[LAPACK] `bool inplace_invert_symmetric(vec2d&)`
- [LAPACK] `bool solve_symmetric(vec2d a, vec1d b, vec1d& r)`
[LAPACK] `bool inplace_solve_symmetric(vec2d& a, vec1d& b)`
- [LAPACK] `bool eigen_symmetric(vec2d a, vec1d& va, vec1d& ve)`
[LAPACK] `bool inplace_eigen_symmetric(vec2d& a, vec1d& va)`
- [FFTW] `vec2cd fft(vec2d)`
[FFTW] `vec2d ifft(vec2cd)`
- `vec_t<1,W> convolve(vec_t<1,T> x, vec_t<1,U> y, vec_t<1,V> k)`

3.8.8 Fitting

- **auto** linfit(**vec_t** y, **vec_t** e, ...)
 auto linfit_pack(**vec_t**<D,T> y, **vec_t**<D,U> e, **vec_t**<D+1,V> x)
- **auto** affinefit(**vec_t**<D,T> y, **vec_t**<D,U> e, **vec_t**<D,V> x)
- **auto** mpfit(F d, vec1d p, **auto** opt = **default**)
- **auto** mpfitfun(**vec_t**<D,T> y, **vec_t**<D,U> e, **vec_t**<D,V> x, F f, vec1d p, **auto** opt = **default**)

3.8.9 Geometry

- vec1u convex_hull(**vec_t**<D,T1> x, **vec_t**<D,T2> y)
- **bool** is_hull_closed(vec1u h, **vec_t**<D,T1> hx, **vec_t**<D,T2> hy)
- **bool** in_convex_hull(T1 x, T2 y, vec1u h, **vec_t**<D,U1> hx, **vec_t**<D,U2> hy)

 vec_t<D,**bool**> in_convex_hull(**vec_t**<D,T1> x, **vec_t**<D,T2> y, vec1u h, **vec_t**<D,U1> hx, **vec_t**<D,U2> hy)
- **double** convex_hull_distance(T1 x, T2 y, vec1u h, **vec_t**<D,U1> hx, **vec_t**<D,U2> hy)

 vec_t<D,**double**> convex_hull_distance(**vec_t**<D,T1> x, **vec_t**<D,T2> y, vec1u h, **vec_t**<D,U1> hx, **vec_t**<D,U2> hy)
- **double** angdistr(**double** ra1, dec1, ra2, dec2)
 double angdist(**double** ra1, dec1, ra2, dec2)
 vec_t<D,**double**> angdist(**vec_t**<D,**double**> ra1, dec1, ra2, dec2)
 vec_t<D,**double**> angdist(**vec_t**<D,**double**> ra1, dec1, **double** ra2, dec2)
- **vec_t**<D,**bool**> angdist_less(**vec_t**<D,**double**> ra1, dec1, **double** ra2, dec2)
- **void** move_ra_dec(**double**& ra, dec, **double** dra, ddec)
 void move_ra_dec(**vec_t**<D,**double**>& ra, dec, **double** dra, ddec)

3.8.10 Debug functions

- `void data_info(vec_t)`
- `void mprint(vec_t<2,T>)`

3.9 Parallel execution

- `void fork(string)`
`void spawn(string)`
- `auto thread::pool(uint_t)`
- `void thread::sleep_for(double)`

3.10 FITS input/output

The FITS (Flexible Image Transport System) format is a general purpose file format developed for astrophysics data. In particular, FITS files can store images with floating point pixel values, image cubes, but also binary data tables with an arbitrary number of columns and rows. Using a meta-data system (FITS keywords), FITS files usually carry a number of important additional informations about their content. E.g., for images files, the mapping between image pixels and sky coordinates (WCS coordinates), or the physical unit of the pixel values.

Storing data tables in binary inside FITS files is a space-efficient and fast way to store and read non-image data. FITS tables come in two fashions: row-oriented and column-oriented tables. In row-oriented tables, all the data about one row (e.g., about one galaxy in the table) is stored contiguously on disk. This means that it is very fast to retrieve all the information about a given object. In column-oriented tables however, a whole column is stored contiguously in memory. This means that it is very fast to read a given column for all the objects in the table. Due to the way the `phy++` library is designed, it is more efficient to use the latter format, since a given column of the file will be represented by a single `phy++` vector. It also brings the nice advantage of allowing to store columns of different lengths, e.g. to combine two tables in the same file, or to carry meta-data that would be hard to store in the standard FITS keywords. The column-oriented format is not well known, but most softwares and libraries do support it¹.

¹Topcat does. In IDL, column-oriented FITS files are supported by the `mrdfits` and `mwrfits` procedures.

Finally, note that this support library introduces a new type: `fits::header` (that we will shorten to `header` in the following). This type is used to store the header of any FITS file. For now, it is actually just a raw `std::string`, but that might change in the future.

We now describe the various functions offered by this library, split into categories.

3.10.1 Generic header functions

- `header fits::read_header(string f)`
`header fits::read_header(string f, uint_t s)`
- **bool** `fits::getkey(header hdr, string k, T& v)`
- **bool** `fits::setkey(header hdr, string k, T& v, string c = "")`

3.10.2 FITS images

- **uint_t** `fits::file_axes(string)`
- `vec1u fits::file_dimensions(string)`
- **bool** `fits::is_cube(string)`
- **bool** `fits::is_image(string)`
- **void** `fits::read(string f, vec_t& v)`
void `fits::read(string f, vec_t& v, header& hdr)`
void `fits::read_hdu(string f, vec_t& v, uint_t hdu)`
void `fits::read_hdu(string f, vec_t& v, uint_t hdu, header& hdr)`
- `vec1s fits::read_sectfits(string f)`
- **void** `fits::write(string f, vec_t v)`
void `fits::write(string f, vec_t v, header hdr)`
- **void** `fits::display(string f)`
void `fits::display(string r, string g)`
void `fits::display(string r, string g, string b)`

3.10.3 WCS coordinates

- **bool** fits::make_wcs_header(**auto** p, header& hdr)
bool fits::make_wcs_header(vec1s p, header& hdr)
- header fits::filter_wcs(header)
- **auto** fits::extast(header)
- **void** fits::ad2xy(**auto** wcs, **vec_t** ra, dec, **vec_t**& x, y)
void fits::xy2ad(**auto** wcs, **vec_t** ra, dec, **vec_t**& x, y)
- **bool** fits::get_pixel_size(string file, **double**& a)

3.10.4 FITS tables

- **vec_t**<1,**auto**> fits::read_table_columns(string)
- **void** fits::read_table(string f, ...)
void fits::read_table_loose(string f, ...)
- **void** fits::write_table(string f, ...)
void fits::update_table(string f, ...)

3.11 Image processing

- **vec_t**<2,T> enlarge(**vec_t**<2,T> v, array {l1, l2, u1, u2}, T d = 0)
vec_t<2,T> enlarge(**vec_t**<2,T> v, **uint_t** u, T d = 0)
- **vec_t**<2,T> shrink(**vec_t**<2,T> v, array {l1, l2, u1, u2})
vec_t<2,T> shrink(**vec_t**<2,T> v, **uint_t** u)
- **void** subregion(**vec_t**<2,T> v, vec1i r, vec1u& rr, vec1u& rs)
void subregion(**vec_t**<2,T> v, vec1i r, T d = 0)
- **vec_t**<2,T> translate(**vec_t**<2,T> v, **double** x, **double** y, T d = 0)
- vec2d circular_mask(vec1u d, vec1d c, **double** r)

- `vec_t<1,T> radial_profile(vec_t<2,T> m, uint_t n)`
- `vec_t<2,T> generate_img(array {w,h}, F f)`
- `vec2d gaussian_profile(array {w,h}, double sigma)`
- `vec_t<2,T> convolve2d(vec_t<2,T> m, vec_t<2,U> k)`
- `vec_t<2,T> boxcar(vec_t<2,T> m, uint_t n, F f)`

3.12 Astrophysics

3.12.1 PSF fitting

- `bool make_psf(array<uint_t> {w, h}, double x0, y0, string model, vec2d& psf)`
- `auto psffit(vec_t<2,T> m, vec_t<2,U> e, vec_t<2,V> psf, vec1i pos)`
`auto psffit(vec_t<2,T> m, vec_t<2,U> e, vec_t<2,V> psf)`

3.12.2 Cosmology

- `auto get_cosmo()`
`auto cosmo_wmap()`
`auto cosmo_std()`
- \mathbb{V} `T lumdist(T z, auto cosmo)`
- \mathbb{V} `T lookback_time(T z, auto cosmo)`
- \mathbb{V} `T vuniverse(T z, auto cosmo)`

3.12.3 Fluxes, magnitudes and luminosities

- `X lsun2uJy(T z, U d, V lam, W lum)`
`X uJy2lsun(T z, U d, V lam, W lum)`
- `W lsun2mag(T lam, U lum, double zp = 23.9)`
`W mag2lsun(T lam, U lum, double zp = 23.9)`

- `U uJy2mag(T flx, double zp = 23.9)`
`U mag2uJy(T mag, double zp = 23.9)`
- `auto read_filter_db(string)`
- `auto get_filter(auto db, string name)`
`auto get_filters(auto db, vec1s names)`
- `double sed2flux(auto fil, vec_t<1,T> lam, vec_t<1,U> sed)`
`vec1d sed2flux(auto fil, vec_t<2,T> lam, vec_t<2,U> sed)`
- `double sed_convert(auto from, auto to, double z, d, vec_t<1,T> lam, vec_t<1,U> sed)`
- `double lir_8_1000(vec_t<1,T> lam, vec_t<1,U> sed)`

3.12.4 Sky coordinates

- `double field_area_hull(vec1u hull, vec_t<D,double> ra, dec)`
`double field_area_hull(vec_t<D,double> ra, dec)`
- `double field_area_h2d(vec_t<D,double> ra, dec)`
- `double field_area(vec_t<D,double> ra, dec)`
- `vec1d angcorrel(vec_t<D1,double> ra, dec, vec_t<D2,double> rra, rdec, vec_t<2,T> b)`
- `auto randpos_uniform(auto seed, vec1d rra, rdec, F in, vec_t<D,double>& ra, dec, auto r)`
- `auto randpos_power_circle(auto seed, double ra0, dec0, r0, vec_t<D,double>& ra, dec, auto r)`
- `auto randpos_power(auto seed, vec1u h, vec_t<D1,double> hra, hdec, vec_t<D2,double>& rra, rdec, auto r)`
- \textcircled{V} `bool sex2deg(string sra, sdec, double& ra, dec)`
- \textcircled{V} `void deg2sex(double ra, dec, string& sra, sdec)`
- \textcircled{V} `auto qxmatch(vec1d ra1, dec1, ra2, dec2, auto options = default)`
- \textcircled{V} `vec2d qdist(vec1d ra, dec, auto options = default)`

3.12.5 Catalog management

- `bool` get_band(`auto` cat, string band, `uint_t`& b)
- `bool` get_band(`auto` cat, string band, string note, `uint_t`& b)

3.12.6 Image stacking

- `bool` pick_sources(`vec_t`<2,T> m, vec1d x, y, `uint_t` hs, `vec_t`<3,T>& c, vec1u& ids)
- `auto` qstack(vec1d ra, dec, string ff, `uint_t` hs, `vec_t`<3,T>& fc, vec1u& i, `auto` option)
- `auto` qstack(vec1d ra, dec, string ff, fw, `uint_t` hs, `vec_t`<3,T>& fc, wc, vec1u& i, `auto` option)
- `auto` qstack_mean(`vec_t`<3,T> fc)
- `auto` qstack_mean(`vec_t`<3,T> fc, wc)
- `auto` qstack_median(`vec_t`<3,T> fc)
- `void` qstack_bootstrap(`vec_t`<3,T> fc, `uint_t` nb, ns, `auto` seed, F f)
- `void` qstack_bootstrap(`vec_t`<3,T> fc, wc, `uint_t` nb, ns, `auto` seed, F f)
- `void` qstack_bootstrap(`uint_t` nb, ns, `auto` seed, F f, ...)
- `vec_t`<3,T> qstack_mean_bootstrap(`vec_t`<3,T> fc, `uint_t` nb, ns, `auto` seed)
- `vec_t`<3,T> qstack_mean_bootstrap(`vec_t`<3,T> fc, wc, `uint_t` nb, ns, `auto` seed)
- `vec_t`<3,T> qstack_median_bootstrap(`vec_t`<3,T> fc, `uint_t` nb, ns, `auto` seed)

3.12.7 Template fitting

- vec2d template_observed(`auto` lib, `double` z, d, `auto` filters)
- vec2d template_observed(`auto` lib, vec1d z, d, `auto` filters)
- \mathcal{V} `double` limweight(`double`)
- `auto` template_fit(`auto` lib, `auto` seed, T z, d, vec1d flux, err, `auto` filters, `auto` opt)

Chapter 4

Tools

4.1 Astrophysics

4.1.1 `angcorrel`

4.1.2 `catinfo`

4.1.3 `deg2sex` and `sex2deg`

4.1.4 `findsrc`

4.1.5 `fluxcube`

4.1.6 `getgal`

4.1.7 `photinfo`

4.1.8 `psffit`

4.1.9 `qstack2`

4.1.10 `qxmacth2`

4.1.11 `randsrc`

4.1.12 `subsrc`

4.2 FITS and ASCII

4.2.1 `fits2ascii`

59

4.2.2 `fitstool`

4.2.3 `imgtool`

4.2.4 `qconvol`

4.2.5 `remcol`

Index

[acos](#), 46
[acosh](#), 46
[affinefit](#), 51
[align_center](#), 44
[align_left](#), 44
[align_right](#), 44
[angcorrel](#), 56, 59
[angdist](#), 51
[angdist_less](#), 51
[angdistr](#), 51
[append](#), 40
[asin](#), 46
[asinh](#), 46
[astar_find](#), 38
[atan](#), 46
[atanh](#), 46

[bessel_i0](#), 46
[bessel_i1](#), 46
[bessel_j0](#), 46
[bessel_j1](#), 46
[bessel_k0](#), 46
[bessel_k1](#), 46
[bessel_y0](#), 46
[bessel_y1](#), 46
[bilinear](#), 49
[bin_center](#), 47
[bin_width](#), 47
[bounds](#), 35

[boxcar](#), 55

[catinfo](#), 59
[ceil](#), 46
[circular_mask](#), 54
[clamp](#), 46
[collapse](#), 43
[complement](#), 34
[convex_hull](#), 51
[convex_hull_distance](#), 51
[convolve](#), 50
[convolve2d](#), 55
[cos](#), 46
[cosh](#), 46
[cosmo_std](#), 55
[cosmo_wmap](#), 55
[count](#), 48
[cut](#), 43

[data_info](#), 52
[deg2sex](#), 56, 59
[derivate1](#), 49
[derivate2](#), 49
[diagonal](#), 50
[dindgen](#), 32
[dinf](#), 46
[distance](#), 44
[dnan](#), 45
[dpi](#), 45

- e10, [46](#)
- eigen_symmetric, [50](#)
- empty, [43](#)
- end_with, [44](#)
- enlarge, [54](#)
- equal_range, [35](#)
- erase_begin, [44](#)
- erase_end, [44](#)
- erf, [46](#)
- erfc, [46](#)
- error, [44](#)
- exp, [46](#)
- fabs, [46](#)
- fft, [50](#)
- field_area, [56](#)
- field_area_h2d, [56](#)
- field_area_hull, [56](#)
- file::directorize, [42](#)
- file::exists, [42](#)
- file::get_basename, [42](#)
- file::get_directory, [42](#)
- file::is_older, [42](#)
- file::list_directories, [42](#)
- file::list_files, [42](#)
- file::mkdir, [42](#)
- file::read_table, [42](#)
- file::write_table, [43](#)
- file::write_table_csv, [43](#)
- file::write_table_hdr, [43](#)
- find, [43](#)
- findgen, [32](#)
- findsrc, [59](#)
- finf, [46](#)
- finite, [46](#)
- fits2ascii, [59](#)
- fits::ad2xy, [54](#)
- fits::display, [53](#)
- fits::extast, [54](#)
- fits::file_axes, [53](#)
- fits::file_dimensions, [53](#)
- fits::filter_wcs, [54](#)
- fits::get_pixel_size, [54](#)
- fits::getkey, [53](#)
- fits::is_cube, [53](#)
- fits::is_image, [53](#)
- fits::make_wcs_header, [54](#)
- fits::read, [53](#)
- fits::read_hdu, [53](#)
- fits::read_header, [53](#)
- fits::read_sectfits, [53](#)
- fits::read_table, [54](#)
- fits::read_table_columns, [54](#)
- fits::read_table_loose, [54](#)
- fits::setkey, [53](#)
- fits::update_table, [54](#)
- fits::write, [53](#)
- fits::write_table, [54](#)
- fits::xy2ad, [54](#)
- fitstool, [59](#)
- flat_id, [31](#)
- flatten, [38](#)
- floor, [46](#)
- fluxcube, [59](#)
- fnan, [45](#)
- fork, [52](#)
- fpi, [45](#)
- fraction_of, [48](#)
- from_string, [43](#)
- gaussian_profile, [55](#)
- generate_img, [55](#)
- get_band, [57](#)
- get_cosmo, [55](#)
- get_filter, [56](#)
- get_filters, [56](#)
- getgal, [59](#)

histogram, [49](#)
histogram2d, [49](#)

identity_matrix, [50](#)
ifft, [50](#)
imgtool, [59](#)
in_bin, [47](#)
in_bin_open, [47](#)
in_convex_hull, [51](#)
indgen, [32](#)
inplace_eigen_symmetric, [50](#)
inplace_invert, [50](#)
inplace_invert_symmetric, [50](#)
inplace_median, [48](#)
inplace_remove, [40](#)
inplace_shuffle, [47](#)
inplace_solve_symmetric, [50](#)
inplace_sort, [32](#)
integrate, [49](#)
integrate_trap, [49](#)
interpolate, [49](#)
invert, [50](#)
invert_symmetric, [50](#)
invsqr, [46](#)
is_any_of, [36](#)
is_hull_closed, [51](#)
is_sorted, [33](#)

keep_end, [44](#)
keep_start, [44](#)

length, [43](#)
limweight, [57](#)
linfit, [51](#)
linfit_pack, [51](#)
lir_8_1000, [56](#)
log, [46](#)
log10, [46](#)
log2, [46](#)

lookback_time, [55](#)
lower_bound, [35](#)
lsun2mag, [55](#)
lsun2uJy, [55](#)
lumdist, [55](#)

mad_clip, [49](#)
mag2lsun, [55](#)
mag2uJy, [56](#)
make_bins, [47](#)
make_psf, [55](#)
make_seed, [47](#)
match, [37](#), [44](#)
match_any_of, [44](#)
match_dictionary, [37](#)
max, [48](#)
max_id, [48](#)
mean, [48](#)
median, [48](#)
min, [48](#)
min_id, [48](#)
mmul, [50](#)
move_ra_dec, [51](#)
mpfit, [51](#)
mpfitfun, [51](#)
mprint, [52](#)
mult_ids, [31](#)

nan, [46](#)
note, [44](#)
now, [45](#)
npos, [30](#)

partial_count, [48](#)
partial_fraction_of, [48](#)
partial_max, [48](#)
partial_mean, [48](#)
partial_median, [48](#)
partial_min, [48](#)

partial_percentile, 48
 partial_total, 48
 percentile, 48
 percentiles, 48
 photinfo, 59
 phypp_check, 41
 pick_sources, 57
 point2d, 50
 pow, 46
 prepend, 40
 print, 44
 print_progress, 45
 progress, 45
 progress_start, 45
 psffit, 55, 59

 qconvol, 59
 qdist, 56
 qstack, 57
 qstack2, 59
 qstack_bootstrap, 57
 qstack_mean, 57
 qstack_mean_bootstrap, 57
 qstack_median, 57
 qstack_median_bootstrap, 57
 qxmatch, 56
 qxmatch2, 59

 radial_profile, 55
 random_coin, 47
 random_pdf, 47
 randomi, 47
 randomn, 47
 randpos_power, 56
 randpos_power_circle, 56
 randpos_uniform, 56
 randsrc, 59
 range, 30
 read_args, 42

 read_filter_db, 56
 rebin, 49
 reform, 39
 remcol, 59
 remove, 40
 remove_extension, 44
 replace, 43
 replicate, 39
 reverse, 33
 rgen, 47
 rgen_log, 47
 rotation_matrix, 50
 round, 46
 run_dim, 48
 run_dim_idx, 48

 scale_matrix, 50
 seconds_str, 45
 sed2flux, 56
 sed_convert, 56
 sex2deg, 56, 59
 shift, 33
 shrink, 54
 shuffle, 47
 sigma_clip, 49
 sign, 46
 sin, 46
 sinh, 46
 solve_symmetric, 50
 sort, 32
 spawn, 52
 split, 43
 sqr, 46
 sqrt, 46
 start_with, 44
 strn, 43
 strn_sci, 43
 strna, 43
 strna_sci, 43

subregion, [54](#)
subsrc, [59](#)
system_var, [44](#)

tan, [46](#)
tanh, [46](#)
template_fit, [57](#)
template_observed, [57](#)
tgamma, [46](#)
thread::pool, [52](#)
thread::sleep_for, [52](#)
time_str, [45](#)
today, [45](#)
tolower, [43](#)
total, [48](#)
toupper, [43](#)

translate, [54](#)
translation_matrix, [50](#)
transpose, [50](#)
trim, [43](#)

uindgen, [32](#)
uJy2lsun, [55](#)
uJy2mag, [56](#)
uniq, [36](#)
upper_bound, [35](#)

vuniverse, [55](#)

warning, [44](#)
where, [34](#)
wrap, [43](#)